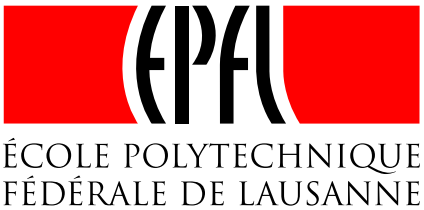
Mining Co-Change Clusters from Version Repositories

Dirk Beyer Andreas Noack



Technical Report No. IC/2005/003
January 26, 2005

Ecole Polytechnique Fédérale de Lausanne
Faculté Informatique & Communications
CH-1015 Lausanne, Switzerland

Mining Co-Change Clusters from Version Repositories

Dirk Beyer
Laboratoire de Modèles et Théorie de Calculs
Faculté Informatique & Communications
Ecole Polytechnique Fédérale de Lausanne
CH-1015 Lausanne, Switzerland
dirk.beyer@epfl.ch

Andreas Noack
Lehrstuhl Software-Systemtechnik
Institut für Informatik, Fakultät 1
Brandenburgische Technische Universität
D-03013 Cottbus, Germany
an@informatik.tu-cottbus.de

Abstract

Clusters of software artifacts that are frequently changed together are subsystem candidates, because one of the main goals of software design is to make changes local. The contribution of this paper is a visualization-based method that supports the identification of such clusters. First, we define the co-change graph as a simple but powerful model of common changes of software artifacts, and describe how to extract the graph from version control repositories. Second, we introduce an energy model for computing force-directed layouts of co-change graphs. The resulting layouts have a well-defined interpretation in terms of the structure of the visualized graph, and clearly reveal groups of frequently co-changed artifacts. We evaluate our method by comparing the layouts for three example projects with authoritative subsystem decompositions.

Classification: D.2.7 Distribution, Maintenance, and Enhancement – Restructuring, reverse engineering, and reengineering, D.2.7 Distribution, Maintenance, and Enhancement – Version control, G.2.2 Graph Theory, I.5.3 Clustering

Keywords: reverse engineering, program comprehension, software clustering, software visualization, software evolution analysis, force-directed graph layout

1. Introduction

Grouping artifacts that are often changed together into subsystems has significant benefits. Changing a member of such a group is likely to trigger changes in other members, and understanding one member of the group often improves the understanding of other members. When such a group forms a subsystem, changes and comprehension processes are more likely to involve only one or few subsystems, and are thus less expensive and error-prone. The benefits are even larger when program sources are grouped not only with other program sources, but also with documentation, test cases or configuration data.

The importance of decomposition in the comprehension and modification of large software systems has lead to the development of many approaches for the automatic and semi-automatic clustering of software artifacts. These approaches can be classified with respect to two criteria: The underlying *model* of the software system, and the notion of *clusters*.

We introduce a new *model* of software systems which is called *co-change graph*. It is an abstraction of version control repositories. The vertices of the co-change graph are software artifacts (such as files or methods) and change transactions (e.g., commits in terms of *CVS*), and the edges connect the change transactions with their participating artifacts.

Previously, clusterings of software artifacts have been derived from file names [4, 34], directories in the file system [2, 63], tokens occurring in source code and documentation files [34, 47, 48], file ownership [2, 14], and in particular from syntactic relationships like calls or variable references [2, 16, 18, 20, 34, 39, 46, 52, 56, 61, 66]. The change history of the software system, as modeled by the co-change graph, promises to be a valuable complement to these information sources for three reasons. First, past common changes of software artifacts appear to be relevant to assess the probability of future common changes, but are not taken into account by the previous approaches. Second, the co-change graph is not restricted to any particular kind of artifacts, while syntax-based models like call graphs only include program source code. Third, the co-change graph can be extracted efficiently and inexpensively from repositories of version control systems like *CVS*. In contrast, the extraction of syntactic relationships like calls requires advanced tools that may produce considerably varying results [53], or may not be available at all for more exotic programming languages.

We introduce a new *clustering method* for co-change graphs which differs from related approaches mainly in two respects: First, the result of the clustering is not a partition of the graph vertices into several clusters, but a layout of the graph vertices (i.e., positions of the graph vertices in two- or three dimensional space). Second, these layouts have a clear interpretation.

In our layouts of the co-change graph, artifacts that are often changed together are placed closely together, while artifacts that participate in few common change transactions are placed at larger distances. Empirical studies have shown that human viewers indeed interpret a close positioning of vertices in a graph layout as relatedness of the corresponding artifacts [13, 22]. Besides being easily comprehensible, a graph layout has the advantage of containing more information than a partition of the set of artifacts. For example, a graph layout can show that an artifact lies at the center of a cluster, at the border of the entire system, or between two clusters, while a partitioning specifies only the membership of the artifact in a particular cluster.

The requirements for the graph layouts are systematically derived from our intuition of co-change clusters, and the positions of the artifacts in the layouts have a clear interpretation in terms of their common changes. Basically, two groups of artifacts are placed closely to the degree that they were changed together more often than random — a notion of clusters similar to ratio cut graph partitioning [64], which was introduced to software clustering by Mancoridis et al. [49].

Our model of co-change in software systems and our clustering method are detailed in Sections 2 and 3. Section 4 evaluates the approach by reporting the results of its application to three software systems. Related work on mining version repositories and clustering is discussed in Section 5.

2. The Co-Change Graph

This section introduces the co-change graph, our model for common changes of software artifacts in version repositories. Its vertices are software artifacts and change transactions, and its edges connect the change transactions with their participating artifacts. The co-change graph can be easily extracted from version repositories. Its simplicity and its clear correspondence to the modeled software system ensure the interpretability of results of its analysis, i.e., valid and efficient inferences of properties of the modeled software system from (probably automatically determined) properties of the model.

After the definition of the co-change graph in the first subsection, the next two subsections discuss design considerations and extensions. The last subsection describes the process of extracting the co-change graph from repositories of the version control system *CVS*.

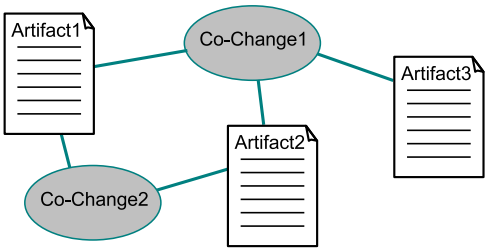


Figure 1. Example co-change graph

2.1. Definition

A *software artifact* is an entity that belongs to a software system, e.g., a package, a file, a function, a line of code, a piece of documentation, or a test case. A version is the state of a software artifact at a particular point in time. Version control systems like *CVS* (Concurrent Versions System) [17] store versions of software artifacts in a central repository. The users of a version control system modify local copies of the software artifacts, and check-in these changes to the central repository from time to time. A *change transaction* is a coherent sequence of check-ins of several software artifacts. Software artifacts that participate in the same change transaction are *co-changed* (commonly changed). Some version control systems, most notably *CVS*, do not store the information which artifacts were checked-in together. In this case, change transactions have to be recovered using timestamps and other logged data.

The *co-change graph* of a given version repository is an undirected graph (V,E). The set of vertices V of the co-change graph contains all software artifacts and all change transactions of the version repository. The set of edges E contains the undirected edge $\{c,a\}$ if and only if the artifact a was changed by the transaction c.

Note that the co-change graph is bipartite, i.e., it contains no edges that connect two change transactions or two software artifacts. Figure 1 shows an example co-change graph with three artifacts and two change transactions, of which one changed three artifacts, and the other changed two artifacts.

For a vertex v of a co-change graph, the number $|\{u \in V \mid \{u,v\} \in E\}|$ of its adjacent vertices is called the *degree* of v and denoted by $\deg(v)$. For transaction vertices, the degree gives the number of artifacts that participate in the transaction, and for artifacts, the degree gives the number of their changes.

2.2. Variants

In the following we point out and justify two decisions we made in our definition of the co-change graph. The first decision is to give all edges the same weight, and the second decision is to include both artifacts and change transactions into the set of vertices, instead of only artifacts.

2.2.1. Weighted Co-Change Graph The *weighted co-change graph* (V,E,w) is an extension of the co-change graph by a weight function $w : E \to \Re$ (where $\Re$ is the set of real numbers). The weight function assigns to each edge a real number, which can be interpreted as the relative importance of the corresponding change.

The standard co-change graph defined in the previous subsection can be considered as a special weighted co-change graph where every edge has the weight 1. Because every edge corresponds to a change of an artifact, it models that every *change of an artifact is equally important*.

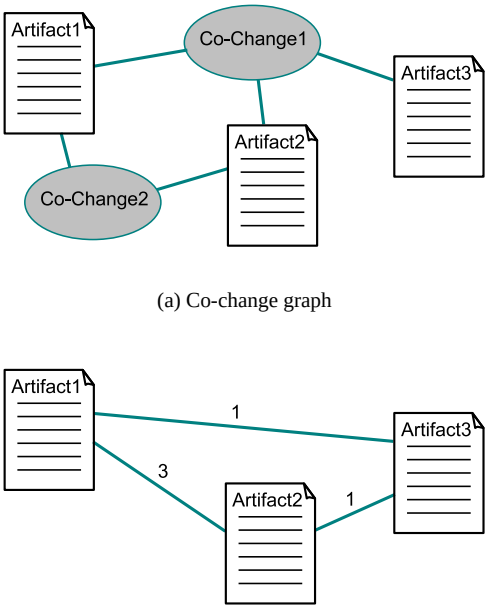
An alternative hypothesis is that every *change transaction is equally important*. This can be modeled by giving each edge to a transaction c the weight $1/\deg(c)$. Then each transaction c contributes $\deg(c)$ edges each with a weight of $1/\deg(c)$, yielding a total weight of 1 for the transaction.

According to our experience and intuition, this model weights co-changes in small change transactions too high. A change transaction of n artifacts corresponds to $\frac{1}{2}n(n-1)$ co-changes of pairs of artifacts. If the transaction has the weight 1, then every co-change has a weight of $\frac{2}{n(n-1)}$. This means, for example, that a co-change in a transaction of 5 files is 19 times as important as a co-change in a transaction of 20 files.

Compromises between the two weighting functions 1 and $1/\deg(c)$, for example $1/\sqrt{\deg(c)}$, make the interpretation of the model more difficult because both the importance of an artifact change and the importance of a change transaction depend on the degree of the change transaction. Thus we prefer the simplest alternative of giving each edge the weight 1.

2.2.2. Condensed Co-Change Graph Because we are mainly interested in co-changes of artifacts, an obvious idea is to remove the transaction vertices from the model, and retain only the artifact vertices. The *condensed co-change graph* for a given version repository is a weighted, undirected graph (V,E,w), where the set of vertices V contains all software artifacts in the repository, the set of edges E contains the edge $\{a,a'\}$ if and only if the artifacts a and a' were commonly changed by a change transaction, and the function $w : E \to \Re$ assigns a weight to each edge.

We have chosen the name 'condensed' to convey the intuition of eliminating the change transaction vertices and the corresponding edges. This loss of information about



(a) Co-change graph

(b) Condensed co-change graph

Figure 2. Example co-change graph and the corresponding condensed co-change graph

change transactions is the obvious disadvantage of the condensed model. It promises to be offset by greater simplicity, but the following discussion shows that the model does not live up to this promise because it requires complicated edge weighting.

Giving each edge the weight 1 does not reflect how often two artifacts were commonly changed. The naive improvement is to weight each edge between two artifacts with the number of times that the artifacts were commonly changed, as done in [24, 31, 67]. But this is also problematic, because it weights large transactions *much* higher than small transactions: Adding a change transaction of n artifacts increases the weights of $\frac{1}{2}n(n-1)$ edges by 1, so the importance of a change transaction is proportional to the square of the number of its participating artifacts. In other words, this weighting scheme (as well as its more complicated variant in [7]) does not reflect that a common change of two artifacts in a small transaction provides more information than a common change of two artifacts in a large transaction.

So we have to include the degree of the change transactions into the weighting function. More precisely, the weight which is added to an edge for a transaction c has to follow a function which is monotonically decreasing in the size $\deg(c)$ of the transaction. The model conforms best to the standard co-change graph if the function is $\frac{2}{\deg(c)-1}$, be-

cause then each transaction c has the weight $\deg(c)$. (It adds $\frac{2}{\deg(c)-1}$ to the weight of $\frac{\deg(c)(\deg(c)-1)}{2}$ edges.)

Figure 2 contrasts a co-change graph (Fig. 2(a)) with the corresponding condensed co-change graph (Fig. 2(b)). Artifact1 and Artifact2 are connected with weight 1, contributed by Co-Change1, and with weight 2, contributed by Co-Change2; the overall weight of the edge is 3. The other edges are weighted with 1, contributed by Co-Change1. The example illustrates that a higher edge weight between two artifacts can be caused by a higher number or by a smaller size of the common change transaction of the two artifacts.

The clustering of the artifacts for the condensed co-change graph with these edge weights is approximately the same as for the standard co-change graph. We prefer the standard co-change graph, because it is simpler (due to the absence of edge weights), and the availability of the transaction vertices improves the traceability of analysis results to the repository.

2.3. Possible Extensions

2.3.1. Low-Level Artifacts Usually, version control systems store artifacts of one particular level of detail (e.g., files for *CVS*). Instead of using this default level, one could extract common changes for artifacts on a lower level of abstraction, such as classes and functions. This can be implemented by analyzing the delta information from the repository (cf. [68, 69] for the mapping of changes to functions). However, this extension depends on the type of the high-level artifacts (program source code, documentation, data), and is therefore not detailed here.

2.3.2. High-Level Artifacts The interpretation of file-level analysis results is tedious for large systems with thousands of files. A co-change graph for high-level artifacts (e.g., directories) can be obtained from a co-change graph for lower-level artifacts (e.g., files) with a transformation called *lifting*: A high-level artifact *a* participates in a change transaction *c* if and only if one of its contained low-level artifacts participates in *c*.

2.3.3. Importance of Changes Additional information about the importance of changes can be included in the co-change graph as edge weights. For example, the intuition that the importance of a change to a file is proportional to the number of changed lines can be formalized by weighting each edge between an artifact and a change transaction with the number of changed lines. Such extensions are promising, but they should be introduced only after the basic model has been evaluated. That they are much simpler to integrate when the model contains transaction vertices is another reason for our decision to prefer the standard co-change graph over the condensed co-change graph.

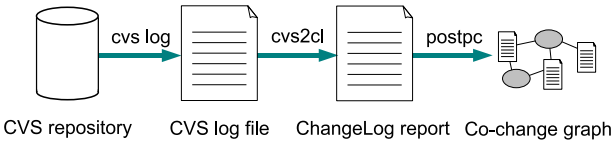


Figure 3. Extraction of the co-change graph from a CVS repository

2.4. Extraction from CVS Repositories

The main problem in the extraction of the co-change graph from *CVS* repositories is that change transactions are not explicitly stored, because *CVS* breaks down its *commit* transactions into sequences of *RCS check-in* operations. However, change transactions can be inferred from available log data with reasonable certainty. In our experiments, we consider a sequence of changes of files as change transaction if the changes have the same user login, the same log message, and timestamps that differ by at most 180 s.

Figure 3 shows a complete tool chain for the extraction of co-change graphs from *CVS* repositories. The command *cvs log* extracts user logins, log messages, and timestamps of changes from the repository, and stores it in a *CVS* log file. The tool *cvs2cl*[1] [8, pp. 247] implements the above heuristic to recover change transactions from the log file, and generates a file in the *GNU ChangeLog* format. Other transaction recovery tools use similar heuristics (e.g., [33, 68, 69]). Finally, a calculator for relations like *CrocoPat* [11] generates the co-change graph from the recovered transactions, and optionally computes edge weights or lifts the graph to obtain more abstract models.

3. Clustering Layout of Co-Change Graphs

Our goal in the analysis of co-change graphs is to identify clusters of artifacts that are frequently changed together. Such groupings can be naturally represented by layouts of the artifacts in two- or three-dimensional space, such that heavily co-changed artifacts are placed closely together, while artifacts that participate in few common change transactions are placed at larger distances.

Energy-based (or force-directed) graph layout methods liken graph vertices to physical objects that exert forces on each other [10, Chapter 10], [15]. Graph vertices that are connected by an edge attract, to ensure that they are placed closely. All pairs of graph vertices repulse, to ensure that non-related vertices are placed at larger distances. The resulting graph layout is an energy-minimal state of the force system.

1 Available at http://www.red-bean.com/cvs2cl

Energy-based graph layout methods have two parts: An energy model which assigns a real number (interpreted as energy) to each graph layout, and an algorithm that searches a layout with minimal energy. There exist several proven solutions for the second aspect, of which we use an efficient algorithm introduced for the simulation of astrophysical systems by Barnes and Hut [9], and first applied for computing graph layouts by Quigley [58].

The contribution of this section concerns the first aspect of energy-based methods. In the first subsection, we systematically derive requirements for the layout of co-change graphs. In the second subsection, we present an energy model whose minimum energy layouts fulfill these requirements, and thus have a clear interpretation in terms of the co-changes of the represented artifacts.

3.1. Requirements for Clustering Layouts of Co-Change Graphs

Intuitively, our requirements for layouts of co-change graphs are small distances between artifacts that participate in many common change transactions, and greater distances between artifacts that participate in few common change transactions. The goal of this subsection is to formalize this intuition.

Consider a co-change graph $G = (V,E)$, and a partition of its set of vertices V into two disjoint sets V_1 and V_2 (i.e., $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$). We require that V_1 and V_2 should be placed closely in the layout to the degree that co-changes between V_1 and V_2 occur more often than random, or equivalently, that their distance is proportional to the degree to which they are co-changed less often than random. More formally, the distance of V_1 and V_2 in the layout should be the quotient of the expected number of edges between V_1 and V_2 in a random graph, and the actual number of edges between V_1 and V_2 in G. (Remember that the edges in the co-change graph represent changes of artifacts, and an edge that connects a vertex in V_1 with a vertex in V_2 represents a change that involves both V_1 and V_2.)

The remainder of this subsection derives a formula for the required distance between V_1 and V_2 from this statement. Therefore, it defines a random graph model, and calculates the expected number of edges between V_1 and V_2 in this random graph model.

First we introduce two notations. The total degree $\sum_{v \in V_i} \deg_G(v)$ of all vertices of V_i in G is denoted by $\deg_G(V_i)$ ($i \in \{1,2\}$). Note that $\deg_G(V_1) + \deg_G(V_2) = 2|E|$. The number of edges $\left|\{\{u,v\} \in E \mid u \in V_1, v \in V_2\}\right|$ between V_1 and V_2 in G is called the *cut* between V_1 and V_2 and denoted by $\mathrm{cut}_G(V_1,V_2)$.

Consider a random graph R with the same set of vertices V and the same number of edges $|E|$ as G, where each of the $2|E|$ end vertices of the edges is randomly chosen from V_1 with the probability $\frac{\deg_G(V_1)}{2|E|}$ and from V_2 with the probability $\frac{\deg_G(V_2)}{2|E|}$. These probabilities are chosen such that the expected total degrees of V_1 and V_2 in R conform to the total degrees in G, namely $\deg_G(V_1)$ and $\deg_G(V_2)$. The expected cut between V_1 and V_2 in R is $\frac{\deg_G(V_1)\deg_G(V_2)}{2|E|}$. So the required distance of V_1 and V_2 in the layout of G, which was defined to be the quotient of this expected cut in the random graph and the actual cut in G, is $\frac{\deg_G(V_1)\deg_G(V_2)}{2|E|\,\mathrm{cut}_G(V_1,V_2)}$.

How are the terms of this formula related to our intuition? Clearly, the distance between V_1 and V_2 should decrease with $\mathrm{cut}_G(V_1,V_2)$, the number of changes involving both V_1 and V_2. However, the same number of common changes (say $\mathrm{cut}_G(V_1,V_2) = 10$) means heavy co-change if V_1 and V_2 are involved in few changes (say $\deg_G(V_1) = \deg_G(V_2) = 20$), but almost complete independence if V_1 and V_2 are involved in a very large number of changes (say $\deg_G(V_1) = \deg_G(V_2) = 2000$). So the distance should indeed be monotonic increasing with $\deg_G(V_1)$ and $\deg_G(V_2)$. The term $2|E|$ in the denominator is constant for a given graph (while the other terms depend on the partition of V into V_1 and V_2), and thus changes only the scaling of the layout.

3.2. The Edge-Repulsion LinLog Energy Model

An energy model specifies what is considered as a good graph layout. It maps graph layouts to real numbers (interpreted as energy) such that smaller numbers mean better layouts. For our visualizations, we use the *edge-repulsion LinLog energy model*:

$$\begin{aligned} U(p) &= \sum_{\{u,v\} \in E} ||p_u - p_v|| \\ &+ \sum_{\{u,v\} \in V^{(2)}} -\deg(u)\deg(v) \ln ||p_u - p_v|| \end{aligned}$$

In this formula, p is a layout (i.e., a mapping of the vertices to positions in two- or three-dimensional space), p_u and p_v are the positions of the vertices u and v in the layout p, and $||p_u - p_v||$ is the Euclidean distance of u and v in p. Remember that $\deg(v)$ is the number of edges of a vertex v.

The first term of the sum can be interpreted as attraction between vertices that are connected by an edge, because its value decreases when the distance of such vertices decreases. The second term can be interpreted as repulsion between all pairs of (different) vertices, because its value decreases when the distance between any two vertices increases. The repulsion of each vertex v is weighted by its number of edges $\deg(v)$. Through this weighting, the second term is more naturally interpreted as repulsion between all pairs of edges than between all pairs of vertices. (More precisely, the repulsion acts not between the entire edges, but only between their end vertices.) So the basic idea behind the edge-repulsion LinLog model is that the edges (in

the co-change graph: changes of artifacts) cause both attraction and repulsion.

For each vertex, the number of attracted vertices is its degree, and its repulsion is weighted with its degree, too. So each vertex has consistently —in terms of attraction and repulsion— an influence on the layout proportional to its degree. This can be visualized by setting the size of a vertex to its degree, as in the figures in Section 4. In the co-change graph, the degree —and thus the importance in the layout— of artifact vertices is the number of change transactions they participate in, and the degree of change transaction vertices is the number of artifacts that participate in the transaction. However, this can be adapted using suitable edge weights, as discussed in the Sections 2.2 and 2.3.

Basically, layouts with minimum edge-repulsion LinLog energy indeed fulfill the requirement identified in the previous subsection, that disjoint sets of vertices V_1 and V_2 have a distance proportional to $\frac{\deg_G(V_1)\deg_G(V_2)}{2|E|\,\mathrm{cut}_G(V_1,V_2)}$. (For a proof and more technical details, we refer to our works on graph layout [55].) However, this holds precisely only for one-dimensional layouts, and only approximately for higher-dimensional (e.g., two-dimensional) layouts. Still, the one-dimensional case is a good approximation when the distances within the sets V_1 and V_2 are small compared to the distance between the sets. Although such an approximate statement about the correspondence between the layout and the analysis goal is not as satisfactory as a precise statement, it is a significant advance over the situation for other energy models, where there are no such statements at all.

4. Evaluation

We evaluate our clustering method by applying it to the CVS repositories of three software systems and comparing the results to authoritative decompositions. The clustering results are layouts —not partitions— which have the disadvantage that similarity measures for partitions (as proposed in [43, 50, 65]) are not applicable, but the advantage that we can present and discuss the results.

The three software systems have different sizes, numbers of developers, and project durations, and include artifacts in various programming languages. Because the evaluation requires the knowledge of good decompositions, we chose systems that we are familiar with. Table 1 gives for each system the overall size (in lines of text), the number of files, the total number of changes of files, the number of commits, the number of users who committed changes, and the project's duration as reflected in the repository. (All numbers were obtained with the tool *StatCvs*[2].)

2 Available at http://statcvs.sourceforge.net

Table 1. Characterization of the example projects

Project	CrocoPat 2.1	Rabbit 2.1	Blast 1.1
Lines	114 000	317 000	3 970 000
Files	60	740	3 900
Changes	800	6300	6800
Commits	140	1 200	900
Users	1	9	8
Months	8	52	40

The co-change graphs were extracted on file level because this enables the application of the same, programming language independent, process and tool chain for all repositories. The layouts of the co-change graphs were computed automatically using the Barnes-Hut algorithm and the edge-repulsion LinLog energy model (introduced in Section 3.2). For comparison, we show layouts obtained with the commonly used Fruchterman-Reingold energy model [29] (discussed in Section 5.2).

The transaction vertices and the edges are elided in the visualizations, and only the artifact vertices are shown, because drawing all edges makes the visualization unreadable. (In an interactive tool they can be shown selectively on demand.) The vertices are displayed as circles, with the area being proportional to the number of transactions the artifact was involved in. Different sizes of corresponding vertices in the Fruchterman-Reingold layouts result from different scaling. (Very small circles were always enlarged to a certain minimum size to ensure their visibility.) The color of the circles reflects the subsystem membership of the corresponding artifact in the authoritative decomposition. Groups of artifacts of the authoritative composition were also (manually) annotated with the names of the subsystems in boxes (gray), to identify them in the text (for grayscale printouts). To avoid overlapping, the names are annotated only for some artifacts. We provide VRML files, which enable navigation through the layouts and contain the complete names of all artifacts, as well as the co-change graphs used for our experiments, on a supplementary web page[3].

The first three subsections of this section evaluate the edge-repulsion LinLog layouts of the three software systems. The fourth subsection shortly discusses the Fruchterman-Reingold layouts.

4.1. CrocoPat 2.1

CrocoPat 2.1 is an interpreter for the language RML (Relational Manipulation Language)[4]. It takes as input an RML program and relations, and outputs resulting relations. The

3 Available at http://mtc.epfl.ch/∼beyer/co-change

4 Available at http://www.software-systemtechnik.de/CrocoPat

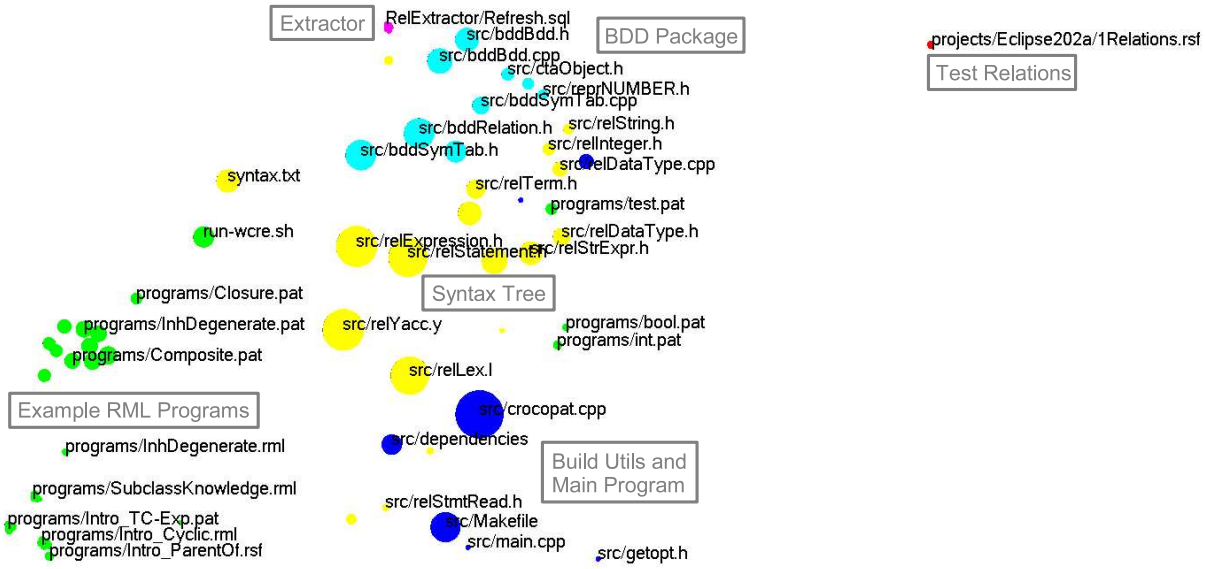


Figure 4. Artifacts in the CrocoPat repository (Edge-repulsion LinLog)

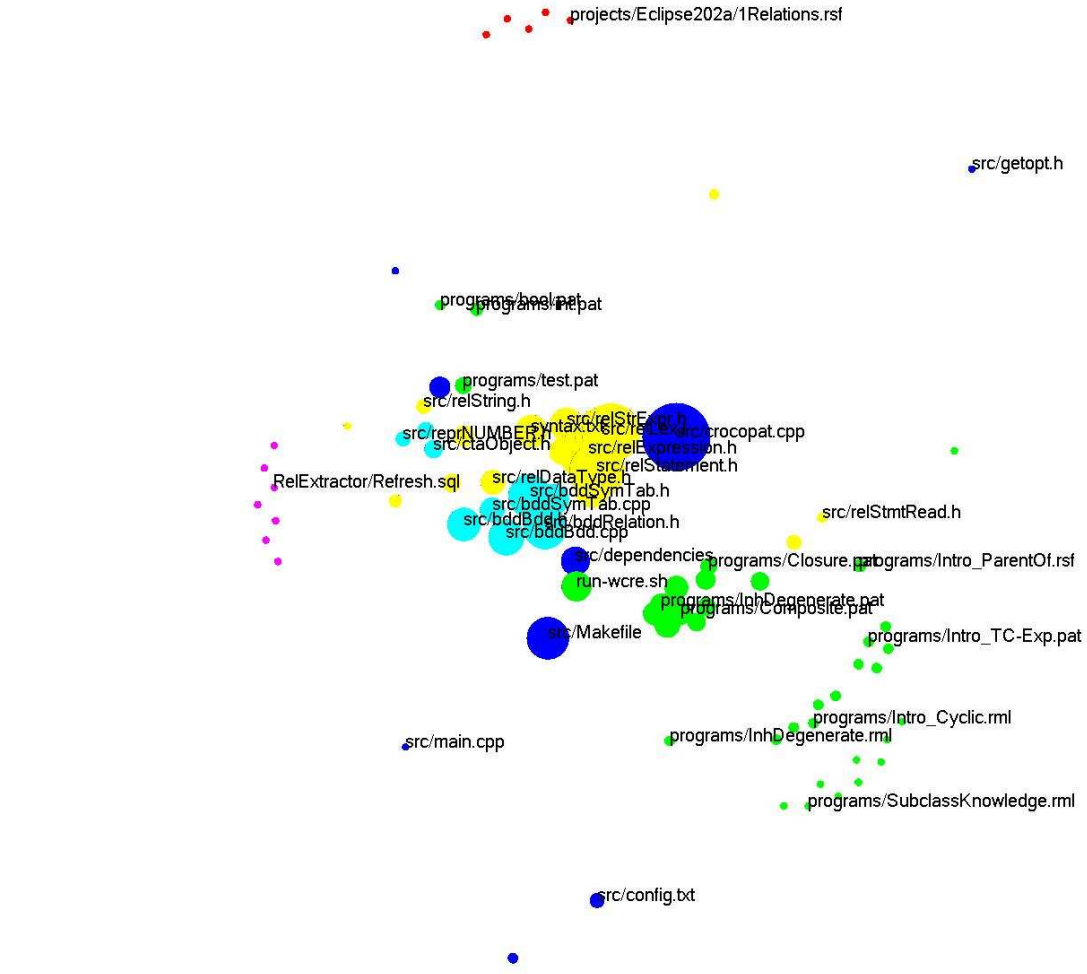


Figure 5. Artifacts in the CrocoPat repository (Fruchterman-Reingold, for comparison)

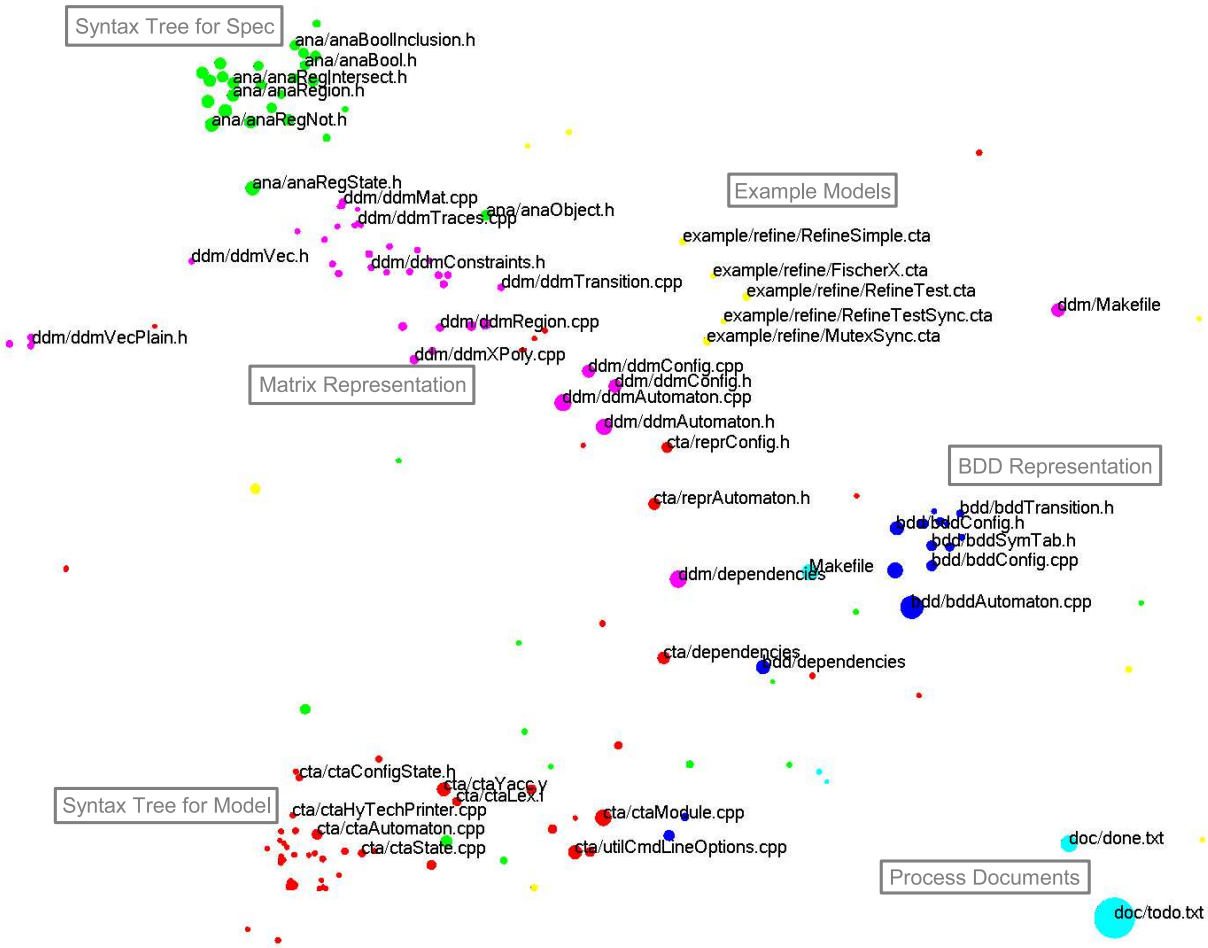


Figure 6. Artifacts in the Rabbit repository (Edge-repulsion LinLog)

repository contains C++ program source code, specifications for the lexical and syntactical analysis of RML programs, SQL scripts, shell scripts, example RML programs, and test relations. It does not include any third party package.

The authoritative decomposition has four major subsystems: program source code, example RML programs (green), test relations (red), and scripts for extracting relations from relational databases (magenta). The program source code subsystem is again divided into three subsystems: build utilities and main program (blue), RML syntax tree (yellow), and BDD package (cyan).

On a global perspective, the layout shows three major clusters of files: The top right cluster contains exactly the test relations (red), the left cluster contains most of the example RML programs (green), and the large central cluster contains the remaining files. We discuss the latter two groups in turn.

The *left cluster* is divided into two subclusters, which belong to two different stable versions of CrocoPat, namely, version 1.3 and version 2.1. A change in the RML syntax between these two versions required changes and renamings in the RML files. The two files `run-wcre.sh` and `syntax.txt` are positioned between the RML programs and program source code for the RML syntax tree. They are indeed related to both subsystems: `run-wcre.sh` is a shell script that runs CrocoPat on some of the old RML programs, and `syntax.txt` is a readable representation of the RML grammar for the tool distribution.

The *large central cluster* contains mainly program source code, but also some other files, which are discussed in the following. A subcluster at the top of the central cluster shows scripts for extracting relations from relational databases (magenta), which were co-changed with the program source code and are thus placed close to it. The layout shows correctly that these scripts belong together, but it does not clearly show that they should be separated from the program source code, to which they are semantically unrelated. The build files (e.g., `dependencies`, `Makefile`) are located at the bottom of the large central cluster. These files are closely related to the program sources, and the authoritative decomposition assigns them to the same subsystem

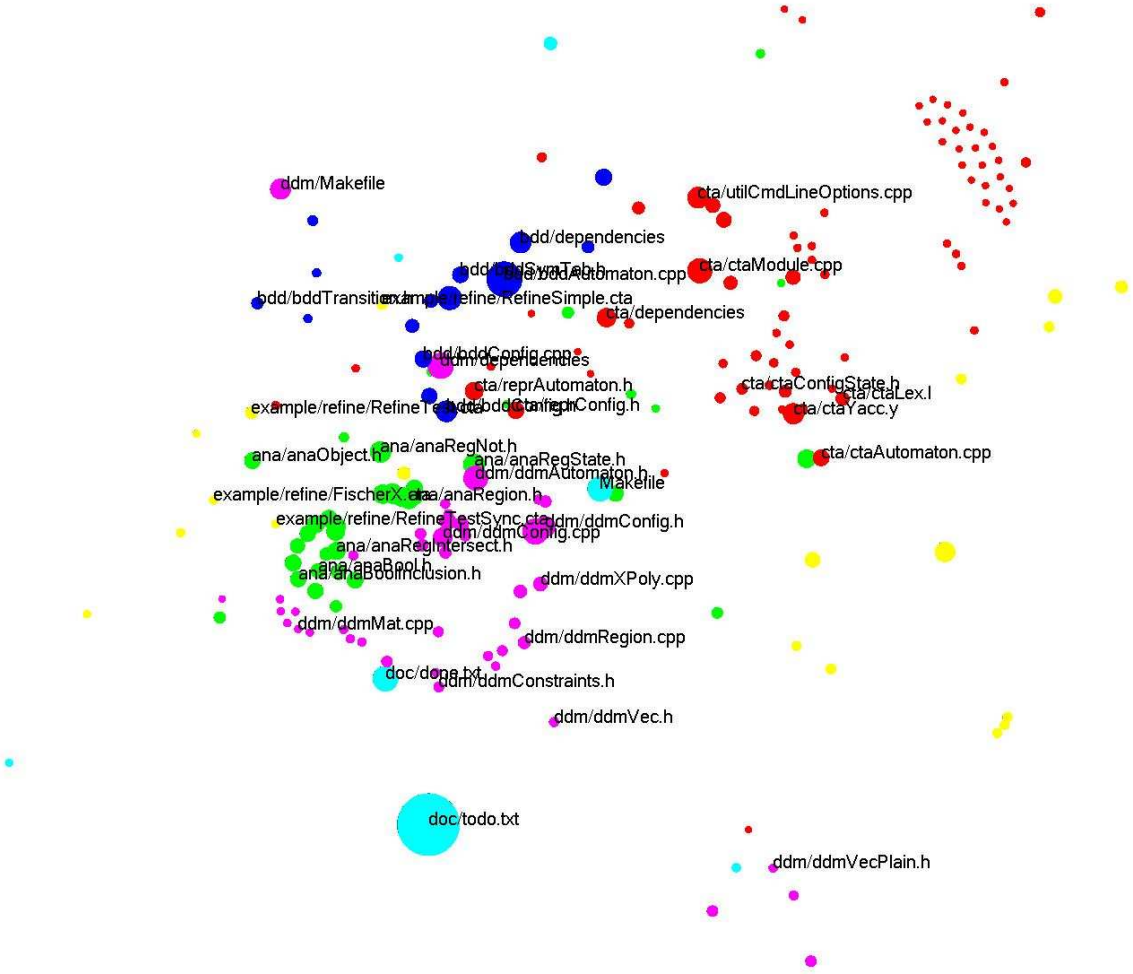


Figure 7. Artifacts in the Rabbit repository (Fruchterman-Reingold, for comparison)

as the main program crocopat.cpp (blue). This is correctly reflected by the layout. Finally, the large central cluster contains three example RML files (test.pat, bool.pat and int.pat, green). The layout suggests to assign these example RML files to the program source subsystem, which differs from the authoritative decomposition, but makes sense, because each of these files is a test case for close program source files.

The *program source code* in the large central cluster is not clearly divided into subclusters, but the placement from bottom to top reflects CrocoPat's layered architecture: The main program crocopat.cpp (blue) starts the RML lexer relLex.l and parser relYacc.y (yellow), the parser builds the RML syntax tree (also yellow), and the syntax tree uses the BDD package (cyan) to calculate with relations.

Besides the interpretation of clusters of files in the layout as subsystem candidates, the positions of files in the layout allow further inferences. For example, the RML parser specification relYacc.y is placed closer to the example RML program files in the left than the main program crocopat.cpp. They are indeed related, because changes of the RML syntax require modifications of both, parser and RML programs. This dependency, as well as several dependencies mentioned earlier, relate artifacts in different languages, and thus could not be detected with syntax-based analyses.

In conclusion, the clustering layout correctly reflects the authoritative decomposition, with two main exceptions. Of these two exceptions, the placement of test cases in the central cluster is semantically justified, but the placement of the database extractor in the central cluster is not. This suggests that historical co-changes should not be over-interpreted for artifacts that were changed rarely (as shown by the small size of the circles).

4.2. Rabbit 2.1

Rabbit 2.1 is a model checking tool for modular timed automata[5]. It is a command line program which takes a

5 Available at http://www.software-systemtechnik.de/Rabbit

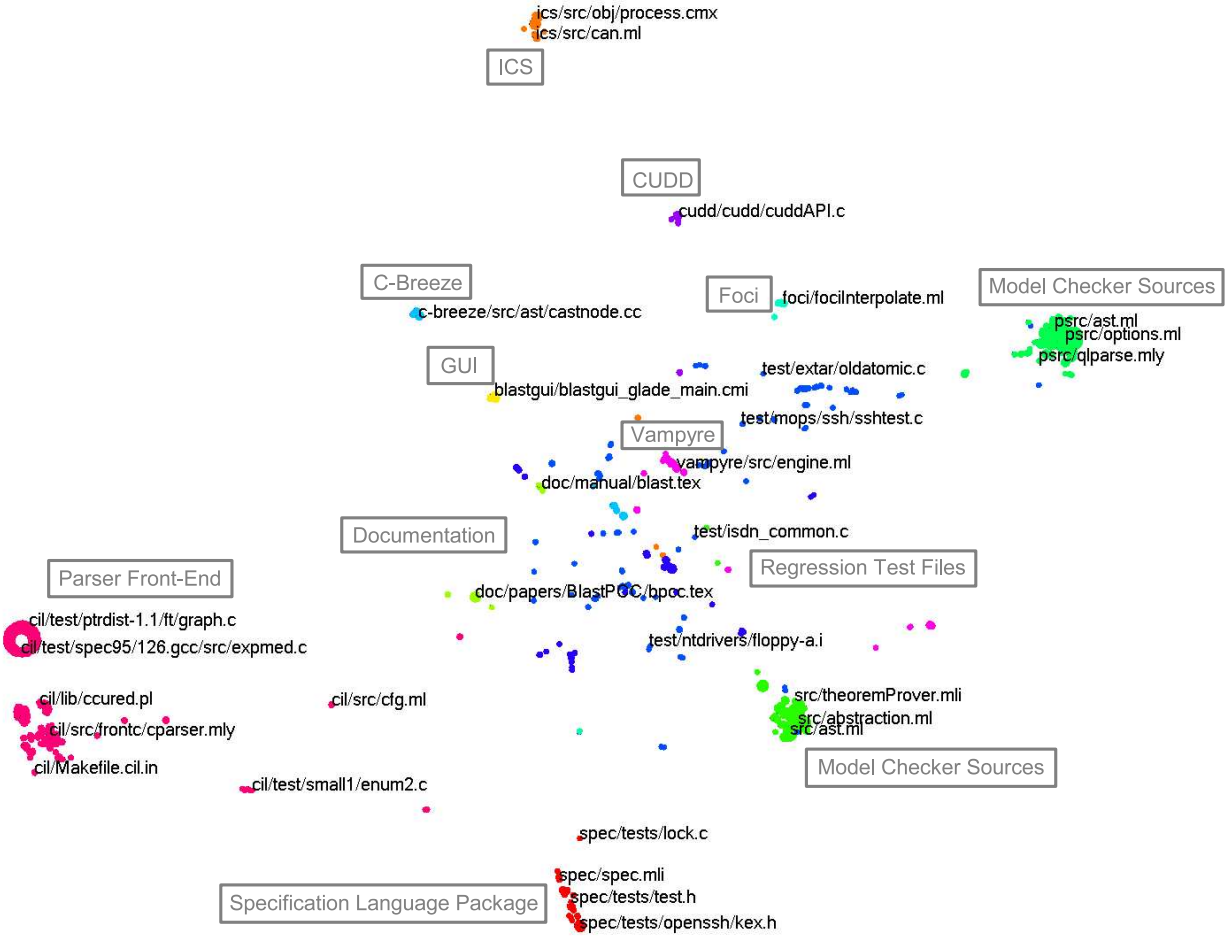


Figure 8. Artifacts in the Blast repository (Edge-repulsion LinLog)

model and specification file as input and writes out verification results. The repository contains C++ code, timed automata models, specification examples, and process documents such as todo and done lists. There is no third party code involved.

The authoritative decomposition has six subsystems, of which the first four contain C++ source code: the syntax tree for specifications (green), the syntax tree for models (red), the matrix representation of models (magenta), the BDD representation of models (blue), example models (yellow), and miscellaneous artifacts including process documents (cyan).

Figure 6 shows only the central part of the layout, some groups of example specifications and models were left out due to space (cf. the supplementary web page for the complete visualization in VRML). The layout correctly groups the four C++ source code subsystems, with some exceptions discussed in the following.

As a first exception, the files reprConfig.h and reprAutomaton.h (both red, center) are placed between the BDD representation cluster (blue) and the matrix representation cluster (magenta), although they belong to the syntax tree for the model in the authoritative decomposition. Here, the placement is correct, and the authoritative decomposition is problematic: The BDD representation and the matrix representation are used alternatively via a common interface, which consists of these two files. In the authoritative decomposition, this common interface could be assigned neither to the BDD subsystem nor to the matrix subsystem, so it was assigned to the even less appropriate syntax tree subsystem.

A second difference between the layout and the authoritative decomposition are build files, for example, the three dependency files (cta/dependencies, bdd/dependencies, and ddm/dependencies). On the one hand, they belong to different source code subsystems, and should be placed closely to the respective clusters. On the other hand, build files are usually changed together, thus should be clustered. The layout reflects these conflicting forces: it places the build files in the center, stretched out to the source code clusters.

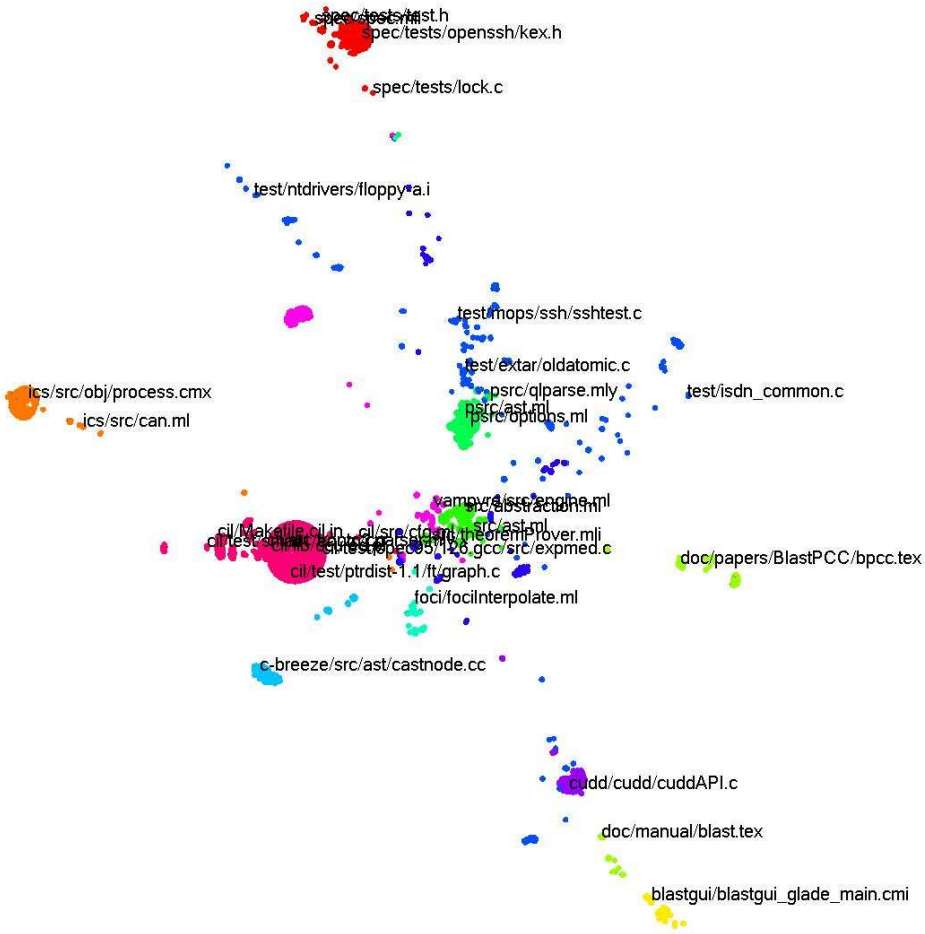


Figure 9. Artifacts in the Blast repository (Fruchterman-Reingold, for comparison)

Besides the representation interface and the build files, some other files of the source code subsystems are placed in the wrong cluster (for example the green files in the red cluster), or outside the main clusters (for example the files around ddmVecPlain.h in the left). As the small size of their representation shows, these files were changed very rarely. For such files, the available co-change information is insufficient to reliably assign them to a subsystem.

Besides the separation of the four main source code subsystems, the layout allows some further inferences about the structure of Rabbit. For example, the magenta cluster of matrix representation code contains two sub-clusters, one top left, and one bottom right. This complicated data structure is indeed divided into a high-level part (automata and configurations) and a low-level part (transition, state, trace, which constitute an automaton, and region, polyhedron, matrix, constraint, which constitute a configuration).

The yellow group of example models (top right) is relatively close to the C++ code. The example models are indeed related to the program files, because they are test cases which were changed together with the tested code. As mentioned earlier, there are other groups of examples, which are not shown because their distances from the central part of the layout are much greater.

The *process documents* (cyan, bottom right) include the project's todo and done list, which are drawn large because they were changed frequently. They were rarely changed together with the source code but mostly in separate reflection phases, as shown by their large distance to the remaining files.

In summary, the main clusters in the layout roughly correspond to Rabbit's actual subsystems. Some clusters are fuzzy and not clearly separated, but Rabbit (like most other real-world software systems) is not composed of perfectly cohesive, mutually independent subsystems, thus clean clusters would not reflect its actual structure.

4.3. Blast 1.1

Blast is a model checker for C programs[6]. It consists of a collection of command line programs and a graphical user interface, and it also includes several third party packages. The repository contains source code in the programming languages Ocaml, C, C++ and Java, regression tests, example C programs, and example specification files.

We can only present an abstract view of the system in Figure 8, due to its considerable size. The figure shows more than 3600 artifacts, and some of the dense groups in the figure consist of several hundred artifacts. A zoom into the groups reveals further details within the subsystems (cf. the supplementary web page for a scalable visualization in VRML).

The authoritative decomposition consists of 12 subsystems. Some of the 12 different colors in Figure 8 are very similar and thus difficult to distinguish.

Four of the main clusters correspond to the four third party packages, namely the C parser front-end Cil with example files (magenta, left), the integrated decision procedure solver package ICS (orange, top), the BDD package CUDD (purple, top), and the compiler infrastructure C-Breeze (light blue, top left). Each of these third party packages was basically (except some configurations and extensions for integration) inserted into the repository in one huge transaction.

Three other large clusters correspond to the actual model checker, split into the current (pscr package, green, right) and an earlier development branch (src package, green, bottom right), and the package for Blast's specification language (spec package, red, bottom).

The central part of the layout shows a cloud of files with some denser accumulations. Three of the accumulations correspond to the Craig interpolation package Foci (cyan, top), Blast's GUI package (yellow, top left), and the proof generating theorem prover Vampyre (magenta, center). The remaining files are documentation (light green, center left) and test cases (blue, center). The widely spread placement of the documentation and test files blurs the separation of the clusters in this area, but is justified because the files are indeed related to several subsystems.

4.4. Fruchterman-Reingold Layouts

The Figures 5, 7 and 9 show the same co-change graphs as the Figures 4, 6 and 8, but the layouts were computed with the Fruchterman-Reingold energy model [29] instead of the edge-repulsion LinLog energy model.

For the purpose of identifying subsystem candidates, the Fruchterman-Reingold layouts have two basic disadvantages. First, they separate clusters less clearly. This is because the Fruchterman-Reingold energy model (as other popular energy models [23, 42, 21]) is not primarily designed for clustering, but for producing readable layouts with uniform edge lengths. Second, the layouts have a strong bias to place nodes with high degree (i.e., artifacts that were involved in many change transactions, and are represented by large circles) in the center and nodes with low degree at the borders. The reason is that nodes with high degree attract more nodes than nodes with low degree, but the repulsion of nodes is independent of their degree. The concept of edge repulsion in the edge-repulsion LinLog energy model avoids this bias.

6 Available at http://www.eecs.berkeley.edu/∼blast

5. Related Work

5.1. Mining Version Repositories

During the last years, mining information from version control repositories has become an increasingly important research area [38]. The aspect of common changes of artifacts has attracted particular attention. The work of Eick and Wills [24] and Ball et al. [7], who visualized a variant of what we call condensed co-change graph, is discussed in Section 2.2. The tool *ROSE* suggests and predicts likely future changes based on the analysis of past co-changes [67, 69]. It provides for a given artifact a prioritized list of other artifacts which are most likely to co-change with it. In contrast, our visualizations are not primarily interpretable with respect to the co-changes of single artifacts, but with respect to more global co-change patterns on the level of groups of artifacts.

Other researchers combine co-changes with related information, such as change reports or problem reports [28], to detect strongly coupled modules [30, 31, 40] or to track features [26, 27]. The latter approach is also related to our work in that it visualizes co-change information, but its visualizations lack interpretability and clarity through the use of a complicated ad hoc weighting scheme and of multi-dimensional scaling for computing layouts (see the discussion in Section 5.2). In the tool Hipikat [19], co-changes are only one of several heuristics used to identify links between all kinds of documents created in a software project, like source code in version control repositories, bug and feature descriptions in bug tracking systems, messages in mailing lists, and design documents on web sites.

Hassan and Holt proposed several heuristics to predict change propagation and a framework to study their performance [37]. In their empirical evaluation, the co-change-based heuristic outperformed the code-structure based heuristic (Call, Use, Define). Sayyad-Shirabad et al. proposed to use classification learning for the generation of prediction models [60]. The method classifies pairs of files

as relevant or not relevant, and the learning algorithm uses syntactic and text-based attributes from source files and associated problem reports. The training examples are classified by past co-changes, i.e., a pair of files is related if it was commonly changed. A relevant pair of the learned relation predicts that a change of one file of the pair propagates a change of the other file.

Hassan and Holt studied changes to derive statements about software complexity [36]. They hypothesize that "a software system becomes complex to manage and maintain when its change history becomes too complex to comprehend". An approach for mining 'frequently applied changes' (FACs) is discussed in [59]. Based on *cvs diff* and clone detection techniques, pieces of code which are often checked-in are mined. It is argued that these code fragments can be general solutions to frequently recurring problems, such as refactoring patterns. Bieman et al. consider frequent changes of a class as indication of architectural importance, but also of chronic problems due to bad design [12]. Mockus and Votta designed a program which automatically classifies maintenance activities to understand reasons for changes [51]. The classification is based on textual descriptions of changes from version repositories. Graves and Mockus used statistical data from version repositories in combination with time sheet data from the financial support system in order to assess predictors for the effort of code changes [35].

Loosely related to the mining of version repositories are pure visualization approaches like [6, 32], which do not aim at detecting hidden patterns in the data themselves, but at amplifying the human abilities to detect such patterns.

5.2. Clustering Techniques

Clustering is the classification of objects into groups based on some notion of similarity or relatedness. Diverse applications have induced the development of many techniques for automatic clustering (see [25, 41] for surveys). In the underlying data model of many clustering techniques, objects are represented as vectors, and their pairwise similarity is specified by a similarity function. In contrast, we model objects as graph vertices and their relationships as graph edges, thus our clustering approach belongs to the subfield of *graph clustering*.

Of the large body of literature on graph clustering (see [1, 57] for surveys), we focus our discussion on work that is related to ours with respect to its three main characteristics: We cluster software artifacts, our notion of clusters is based on cuts, and we compute clusters with energy-based methods.

5.2.1. Software Clustering Almost all techniques for the graph-based clustering of software artifacts have been developed for graph models of the static structure of software systems. That most of these techniques are designed specifically for directed graphs is the lesser problem, because the edges of a co-change graph can easily be defined as directed (e.g., from artifacts to change transactions). The main problem is the reliance on particular semantics of the graph vertices and edges. For example, dominance analysis or the computation of strongly connected components are useful for call graphs [18], but their application to (directed) co-change graphs makes no sense. (All their strongly connected components have exactly one vertex.) More general and potentially applicable to co-change graphs are similarity-based techniques (e.g., [61]) and concept analysis (discussed, e.g., in [3]). The only two software clustering approaches with a significant relation to our work are *Bunch* [49], which belongs to the cut-based graph clustering techniques discussed in the next paragraph, and the work of Eick and Wills [24], who use an ad hoc energy model for weighted graphs to reveal graph clusters.

5.2.2. Normalized Cuts as Graph Clustering Criterion The cut between two disjoint sets of graph vertices is the number of edges that connect both sets (as defined in Section 3.1). Several researchers have proposed the minimization of certain normalized forms of the cut as clustering criterion. The normalization of the cut with the maximum possible number of edges between the two sets of vertices is called the ratio of the cut [64], and is also used in *Bunch* [49]. However, the ratio of the cut is biased when the degrees of the graph vertices are very nonuniform [55]. That is why we chose in Section 3.1 to normalize the cut with the expected number of edges in a random graph model, as done earlier (but without a systematic derivation) in [62].

5.2.3. Energy-Based Graph Clustering Many energy models have appeared in the literature on automatic graph drawing (most prominently, [23, 42, 29, 21]). The goal of these energy models are readable layouts of graphs in a particular drawing convention where edges are drawn as straight lines. This drawing convention is not applicable to typical co-change graphs because they are so dense (see Table 1) that showing all edges creates heavy clutter.

The above mentioned energy models enforce that the edge lengths are as uniform as possible, but this contradicts our goal of separating clusters of strongly connected vertices, which requires some long edges between the clusters and much shorter edges within the clusters (as shown in [54]). All mentioned energy models easily generalize to graphs with weighted edges (in which case they are similar to multidimensional scaling [44]). Given appropriate edge weights, their minimum energy layouts reveal clusters, but this means putting clusters in (in the form of edge weights) to get clusters out. In contrast, the goal of our edge-repulsion LinLog energy model is not purely the represen-

tation of given knowledge in two or three dimensions, but to some degree also the discovery of *new* knowledge.

A distinguishing feature of energy-based clustering compared to other clustering approaches is that it does not produce partitions or dendrograms, but layouts. Layouts have the advantages of being easily comprehensible and containing more information, because they can also show, e.g., that a vertex is rather between two clusters than belonging to one of these clusters, or how clearly two clusters are separated. Theoretical connections between conventional clustering and the creation of layouts show that both approaches are closely related [5, 45].

6. Conclusion

This paper introduced a new method for clustering software artifacts, based on historical co-changes and interpretable graph layout. First, we defined the *co-change graph* as underlying formal model, which has —in comparison to syntax-based models— the advantages of being inexpensively extractable and not restricted to program source code or certain programming languages. Second, we systematically derived requirements for the layout of co-change graphs, and introduced an energy model for computing such layouts. We evaluated the method on three example software systems with different types of documents and source code in several programming languages. The clusters in the resulting layouts basically conformed to the authoritative decompositions of the software systems, and revealed further interesting details that cannot be represented in a pure partitioning. However, the method is not reliable for artifacts that were changed very rarely. For such artifacts, historical common changes have to be combined with other information to improve their assignment to subsystems.

References

[1] C. J. Alpert and A. B. Kahng. Recent directions in netlist partitioning: A survey. *Integration, the VLSI Journal*, 19(1-2):1–81, 1995.

[2] P. Andritsos and V. Tzerpos. Software clustering based on information loss minimization. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, pages 334–344. IEEE Computer Society, 2003.

[3] N. Anquetil. A comparison of graphs of concept for reverse engineering. In *Proceedings of the 8th IEEE International Workshop on Program Comprehension (IWPC 2000)*, pages 231–240. IEEE Computer Society, 2000.

[4] N. Anquetil and T. Lethbridge. Extracting concepts from file names: A new file clustering criterion. In *Proceedings of the 20th International Conference on Software Engineering (ICSE 1998)*, pages 84–93. IEEE Computer Society, 1998.

[5] Y. Aumann and Y. Rabani. An $O(logk)$ approximate min-cut max-flow theorem and approximation algorithm. *SIAM Journal on Computing*, 27(1):291–301, 1998.

[6] T. Ball and S. G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[7] T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy. If your version control system could talk ... In *Proceedings of the ICSE '97 Workshop on Process Modelling and Empirical Studies of Software Engineering*, 1997.

[8] M. Bar and K. Fogel. *Open Software Development with CVS*. Paraglyph Press, Scottsdale (AZ), 3rd edition, 2003.

[9] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324:446–449, 1986.

[10] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Upper Saddle River, NJ, 1999.

[11] D. Beyer and A. Noack. Crocopat 2.1 introduction and reference manual. Technical Report UCB//CSD-04-1338, Computer Science Division (EECS), University of California, Berkeley, 2004.

[12] J. M. Bieman, A. A. Andrews, and H. J. Yang. Understanding change-proneness in OO software through visualization. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC 2003)*, pages 44–53. IEEE Computer Society, 2003.

[13] J. Blythe, C. McGrath, and D. Krackhardt. The effect of graph layout on inference from social network data. In F.-J. Brandenburg, editor, *Proceedings of the Symposium on Graph Drawing (GD 1995)*, LNCS 1027, pages 40–51, Berlin, 1996. Springer-Verlag.

[14] I. T. Bowman and R. C. Holt. Reconstructing ownership architectures to help understand software systems. In *Proceedings of the 7th International Workshop on Program Comprehension (IWPC 1999)*, pages 28–37. IEEE Computer Society, 1999.

[15] U. Brandes. Drawing on physical analogies. In M. Kaufmann and D. Wagner, editors, *Drawing Graphs: Methods and Models*, LNCS 2025, pages 71–86. Springer-Verlag, Berlin, 2001.

[16] G. Canfora, A. Cimitile, and M. Munro. An improved algorithm for identifying objects in code. *Software: Practice and Experience*, 26(1):25–48, 1996.

[17] P. Cederqvist et al. *Version Management with CVS*. Free Software Foundation, 2004.

[18] A. Cimitile and G. Visaggio. Software salvaging and the call dominance tree. *Journal of Systems and Software*, 28(2):117–127, 1995.

[19] D. Cubranic and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 408–418. IEEE Computer Society, 2003.

[20] J. Davey and E. Burd. Evaluating the suitability of data clustering for software remodularization. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE 2000)*, pages 268–276. IEEE Computer Society, 2000.

[21] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15(4):301–331, 1996.

[22] E. Dengler and W. Cowan. Human perception of laid-out graphs. In S. H. Whitesides, editor, *Proceedings of the 6th International Symposium on Graph Drawing (GD 1998)*, LNCS 1547, pages 441–443, Berlin, 1998. Springer-Verlag.

[23] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[24] S. G. Eick and G. J. Wills. Navigating large networks with hierarchies. In *Proceedings of IEEE Visualization 1993*, pages 204–210, 1993.

[25] B. S. Everitt, S. Landau, and M. Leese. *Cluster Analysis*. Hodder Arnold, London, 4th edition, 2001.

[26] M. Fischer and H. Gall. Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):385–403, 2004.

[27] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, pages 90–99. IEEE Computer Society, 2003.

[28] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, pages 23–. IEEE Computer Society, 2003.

[29] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software – Practice and Experience*, 21(11):1129–1164, 1991.

[30] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance (ICSM 1998)*, pages 190–197. IEEE Computer Society, 1998.

[31] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 84–94. IEEE Computer Society, 2003.

[32] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *Proceedings of the International Conference on Software Maintenance (ICSM 1999)*, pages 99–108. IEEE Computer Society, 1999.

[33] D. M. German. Mining CVS repositories, the softChange experience. In *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR 2004)*, pages 17–21, 2004.

[34] J.-F. Girard, R. Koschke, and G. Schied. A metric-based approach to detect abstract data types and state encapsulations. *Automated Software Engineering*, 6(4):357–386, 1999.

[35] T. L. Graves and A. Mockus. Inferring change effort from configuration management databases. In *Proceedings of the 5th International Software Metrics Symposium (METRICS 1998)*, pages 267–273. IEEE Computer Society, 1998.

[36] A. E. Hassan and R. C. Holt. The chaos of software development. In *Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 84–94. IEEE Computer Society, 2003.

[37] A. E. Hassan and R. C. Holt. Predicting change propagation in software systems. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, pages 284–293. IEEE Computer Society, 2004.

[38] A. E. Hassan, R. C. Holt, and A. Mockus. Proceedings of the 1st international workshop on mining software repositories (MSR 2004), 2004.

[39] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, 11(8):749–757, 1985.

[40] J. Itkonen, M. Hillebrand, and V. Lappalainen. Application of relation analysis to a small Java software. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 233–239, 2004.

[41] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[42] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[43] R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proceedings of the 8th IEEE International Workshop on Program Comprehension (IWPC 2000)*, pages 201–210. IEEE Computer Society, 2000.

[44] J. B. Kruskal and M. Wish. *Multidimensional Scaling*. Sage Publications, Beverly Hills, CA, 1978.

[45] N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15(2):215–245, 1995.

[46] P. E. Livadas and T. Johnson. A new approach to finding objects in programs. *Journal of Software Maintenance: Research and Practice*, 6(1):249–260, 1994.

[47] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, 1991.

[48] J. I. Maletic and N. Valluri. Automatic software clustering via latent semantic analysis. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE 1999)*, pages 251–254. IEEE Computer Society, 1999.

[49] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 1999)*, pages 50–59. IEEE Computer Society, 1999.

[50] B. S. Mitchell and S. Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001)*, pages 744–753. IEEE Computer Society, 2001.

[51] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the 26th International Conference on Software Maintenance (ICSM 2000)*, pages 120–130. IEEE Computer Society, 2000.

[52] H. A. Müller and J. S. Uhl. Composing subsystem structures using (k,2)-partite graphs. In *Proceedings of the International Conference on Software Maintenance (ICSM 1990)*, pages 12–19. IEEE Computer Society, 1990.

[53] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S.-C. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, 1998.

[54] A. Noack. An energy model for visual graph clustering. In G. Liotta, editor, *Proceedings of the 11th International Symposium on Graph Drawing (GD 2003)*, LNCS 2912, pages 425–436, Berlin, 2004. Springer-Verlag.

[55] A. Noack. Visual clustering of graphs with nonuniform degrees. Technical Report 02/04, Institute of Computer Science, Brandenburg University of Technology at Cottbus, 2004.

[56] S. Patel, W. Chu, and R. Baxter. A measure for composite module cohesion. In *Proceedings of the 14th International Conference on Software Engineering (ICSE 1992)*, pages 38–48, 1992.

[57] A. Pothen. Graph partitioning algorithms with applications to scientifi c computing. In D. E. Keyes, A. Sameh, and V. Venkatakrishnan, editors, *Parallel Numerical Algorithms*, pages 323–368. Kluwer, 1997.

[58] A. J. Quigley and P. Eades. FADE: Graph drawing, clustering, and visual abstraction. In J. Marks, editor, *Proceedings of the 8th International Symposium on Graph Drawing (GD 2000)*, LNCS 1984, pages 197–210, Berlin, 2001. Springer-Verlag.

[59] F. V. Rysselberghe and S. Demeyer. Mining version control systems for FACs (frequently applied changes). In *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR 2004)*, pages 48–52, 2004.

[60] J. Sayyad-Shirabad, T. Lethbridge, and S. Matwin. Mining the maintenance history of a legacy software system. In *Proceedings of the 26th International Conference on Software Maintenance (ICSM 2003)*, pages 95–104. IEEE Computer Society, 2003.

[61] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering (ICSE 1991)*, pages 83–92. IEEE Computer Society, 1991.

[62] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[63] V. Tzerpos and R. C. Holt. ACDC: An algorithm for comprehension-driven clustering. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE 2000)*, pages 258–267. IEEE Computer Society, 2000.

[64] Y.-C. Wei and C.-K. Cheng. Ratio cut partitioning for hierarchical design. *IEEE Transactions on Computer-Aided Design*, 10(7):911–921, 1991.

[65] Z. Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC 2004)*, pages 194–203. IEEE Computer Society, 2004.

[66] A. S. Yeh, D. R. Harris, and H. B. Reubenstein. Recovering abstract data types and object instances from a conventional procedural language. In *Proceedings of the 2nd Working Conference on Reverse Engineering (WCRE 1995)*, pages 227–236. IEEE Computer Society, 1995.

[67] T. Zimmermann, S. Diehl, and A. Zeller. How history justifi es system architecture (or not). In *Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 73–83. IEEE Computer Society, 2003.

[68] T. Zimmermann and P. Weigerber. Preprocessing CVS data for fi ne-grained analysis. In *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR 2004)*, pages 2–6, 2004.

[69] T. Zimmermann, P. Weigerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572. IEEE Computer Society, 2004.